UCRL- 96034
PREPRINT

CIRCULATION COP
O RECALL
WEEKS

GIGAFLOP SPEED ALGORITHM FOR THE DIRECT SOLUTION OF LARGE BLOCK-TRIDIAGONAL SYSTEMS IN 3D PHYSICS APPLICATIONS

David V. Anderson
Alan E. Fry
Ralf Gruber
Alexandre Roy

This paper was prepared for submittal to
Journal of Parallel Computing

January 1987

Lawrence
Livermore
National
Laboratory

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

January 20, 1987

Gigaflop Speed Algorithm for the Direct Solution of Large Block-Tridiagonal Systems in 3D Physics Applications

David V. Anderson and Alan R. Fry
National Magnetic Fusion Energy Computer Center
Lawrence Livermore National Laboratory
Livermore, California, 94550, USA

Ralf Gruber and Alexandre Roy
Centre de Recherche en Physique des Plasmas
Association Euratom-Confederation Suisse
Ecole Polytechnique Federale de Lausanne
CH-1007 Lausanne, Switzerland

ABSTRACT

In the discretization of the 3D partial differential equations of many physics problems, it is found that the resultant system of linear equations can be represented by a block tridiagonal matrix. Depending on the substructure of the blocks one can devise many algorithms for the solution of these systems. For plasma physics problems of interest to the authors several interesting matrix problems arise which we expect will be useful in other applications as well. In one case, where the blocks are dense, we have found that by using a multitasked cyclic reduction procedure, it is possible to reach gigaflop rates on a Cray-2 for the direct solve of these large linear systems. We have recently built a new code PAMS (PArallelized Matrix Solver) that embodies this technique and uses fast vendor supplied routines and obtains this good performance. Manipulations within the blocks are done by these highly optimized linear algebra subroutines that exploit vectorization as well as overlap of the functional units within each CPU. In unitasking mode, speeds about 340 megaflops have been measured. The cyclic reduction method multitasks quite well with overlap factors in the range of three to four. In multitasking mode, average speeds of 1.1 gigaflops have been measured for the entire PAMS algorithm. In addition to the presentation of the PAMS algorithm, we show how related systems having banded blocks may be treated efficiently by multitasked cyclic reduction in the Cray-2 multiprocessor environment. The PAMS method is intended for multiprocessors and would not be a method of choice on a uniprocessor. Furthermore, we find this method's advantage to be critically dependent on the hardware, software, and charging algorithm installed on any given multiprocessor system.

1. Introduction

The arrival of supercomputers having many processors with a total computing power exceeding 1 gigaflops ($> 10^9$ floating point operations per second) and memories in the range of 64 million to 256 million 64 bit words, forces us to adapt the algorithms of our physics calculations to the architecture of these computers. Depending on the sparseness of the matrix system that arises in a given physical problem one may choose iterative or direct methods of solution. We note that for a large class of problems the direct methods are most efficient, particularly for computers with large memories. In this paper, we present a set of direct matrix solvers specifically adapted to the multiprocessor shared memory supercomputer of which the Cray-2 is representative. We are hopeful these methods will carry over to the new line of ETA multiprocessors and other machines of this type.

Some of the matrix problems presented here are those encountered in some of the authors' calculations of the equilibrium and linear stability properties of magnetically confined plasmas. Two of the three problems presented have analogues in other applications while the third one may be unique in our estimation.

Although the bookkeeping of the algorithms used here is somewhat complicated, the basic idea is simple. These global matrix problems are represented by the two level structure of a big matrix composed of block elements that are in turn familiar elementary matrices. Calculations within blocks will be treated by highly optimized vector coding, while calculations at the global level will be multitasked to the extent possible. In connection with this latter point, the method of cyclic reduction leads to an algorithm that is easily multitasked.

The notation of this paper will use upper case letters to represent global matrices and vectors, while lower case letters will be used to represent the block sub-matrices and associated sub-vectors. We shall have no occasion to refer to the actual elements within these blocks or sub-vectors as these are manipulated by standard techniques of linear algebra with which the reader is assumed to be familiar. The use of positive integer superscripts will refer to the reduction level and the use of the notation b^{-1} will mean the matrix inverse representation of the block b.

We organize this paper as follows. In Section 2 we describe the matrix forms that have dense blocks and show in detail, by way of a specific example, how the PAMS code solves such a problem. Then in Section 3 we describe two other matrix problems that possess structures that require a somewhat different approach at first, but which after one stage of reduction are brought into the form solved in Section 2. In Section 4, we present results of actual Cray-2 runs in which very favorable performance was measured for the entire matrix solver. At a speed of 1.1 gigaflops we believe PAMS may be the fastest application code running on a Cray-2. Section 5 discusses what has been achieved here, describes limitations of the method, and concludes somewhat equivocally as to the value of the PAMS approach relative to various multitasking environments. Finally, appendices present the general formulae of the PAMS method and more details about the sparse techniques of Section 3.

2. The Solution of Tridiagonal Problems with Dense Blocks

Consider the matrix problem $A_1X = Y$ which has the structure shown in Fig. 1. The matrix A_1 is block tridiagonal with dense blocks. In a typical problem there may be on the order of 100 rows of blocks comprising the global matrix. The blocks, in turn, consist of densely filled elementary matrices with rank of order a few hundred. Such matrices arise if, for instance, one expands a 3D partial differential equation using global functions in two coordinates and finite elements in the third one. The number of block rows equals the number of expansion functions for the third coordinate, while the rank of the dense blocks equals the product of the dimensions of the global functions used in the first two coordinates.

We have drawn this example from a 3D toroidal plasma equilibrium calculation which we are currently developing. We are in the process of generalizing the analysis of the ERATO 2D equilibrium and stability methods to three dimensions.[1] The poloidal and toroidal angle coordinates are periodic and are being treated by finite Fourier representations in which each basis function spans the full domain. The radial (toroidal minor radius) coordinate is being represented by local finite element basis functions. The tensor product of the three coordinate representations forms the 3D model in which we expand the equilibrium equations. When the resulting linear equations for the expansion coefficients are ordered such that the indices of the angle coordinates are the fastest turning, then the matrix for this problem is block tridiagonal with dense blocks. Within each block the elements are representing the coupling among the angle coordinates; since the basis functions of the angle coordinates span the full domain, every element is generally non-zero. Thus the blocks are dense. The index that labels the blocks corresponds to the radial coordinate where the basis functions are localized. We are here assuming that only "adjacent" basis functions couple which is typical of linear "tent" functions. This local coupling leads to the tridiagonal form among the blocks. These choices for basis functions have been found to be near optimal for the kinds of toroidal plasma configurations currently of interest in the magnetic fusion research community and have been used in many other theoretical models. For example, the analysis of the PEST code (Princeton Equilibrium and Stability Code) could be reduced to this form A_1.[2]

To make these ideas clear, let us consider the case for $N = 7$ block rows in the global matrix. For the first matrix problem we are considering, the global equation is

$$A_1 \; X = Y.$$

Written out we have 7 block equations

$$\begin{aligned}
&b_1^1 x_1 + c_1^1 x_2 &&= d_1^1 \\
&a_2^1 x_1 + b_2^1 x_2 + c_2^1 x_3 &&= d_2^1 \\
&\quad a_3^1 x_2 + b_3^1 x_3 + c_3^1 x_4 &&= d_3^1 \\
&\qquad a_4^1 x_3 + b_4^1 x_4 + c_4^1 x_5 &&= d_4^1 \\
&\qquad\quad a_5^1 x_4 + b_5^1 x_5 + c_5^1 x_6 &&= d_5^1 \\
&\qquad\qquad a_6^1 x_5 + b_6^1 x_6 + c_6^1 x_7 &&= d_6^1 \\
&\qquad\qquad\quad a_7^1 x_6 + b_7^1 x_7 &&= d_7^1.
\end{aligned} \tag{1}$$

This system of block matrix equations can be solved by cyclic elimination techniques. In words, we use substitution to eliminate all of the odd numbered equations which at the first stage of reduction leads to a global system of rank=3 in terms of the number of blocks. Repeating this procedure again leads to a rank=1 system that we can solve directly using techniques appropriate to dense elementary matrices. Since the formulas of substitution used in the elimination procedure are all saved, one can obtain all of the unknown subvectors x_i from the solution of the rank=1 system.

This process proceeds through several stages labelled by k as follows,

$$k = 1, \ldots, (\log_2 \; N).$$

Within each stage we eliminate all of the x_j given by

$$j = (2i-1)2^{k-1}; \qquad\qquad i = 1, \ldots, \frac{N+1}{2^k}.$$

The notation $(\log_2 N)$ is to be interpreted as meaning the greatest integer bounded by $(\log_2 N)$.

Proceeding with the case at hand, where $N = 7$, we write out the solutions of the odd-numbered sub-vectors by inspection of Eq. 1. That is, at the first level of reduction $k = 1$ and $i = 1, 2, 3, 4$. Thus we eliminate the sub-vectors x_1, x_3, x_5, and x_7 by solving the odd numbered block rows to obtain,

$$\begin{aligned}
x_1 &= (b_1^1)^{-1}(d_1^1 - c_1^1 x_2)\\
x_3 &= (b_3^1)^{-1}(d_3^1 - a_3^1 x_2 - c_3^1 x_4)\\
x_5 &= (b_5^1)^{-1}(d_5^1 - a_5^1 x_4 - c_5^1 x_6)\\
x_7 &= (b_7^1)^{-1}(d_7^1 - a_7^1 x_6).
\end{aligned} \tag{2}$$

By substitution of Eqs. (2) into the remaining, even numbered, block rows we get,

$$\begin{aligned}
b_1^2 x_2 + c_1^2 x_4 &= d_1^2\\
a_2^2 x_2 + b_2^2 x_4 + c_2^2 x_6 &= d_2^2\\
a_3^2 x_4 + b_3^2 x_6 &= d_3^2
\end{aligned} \tag{3}$$

where

$$\begin{aligned}
b_1^2 &= b_2^1 - a_2^1(b_1^1)^{-1}c_1^1 - c_2^1(b_3^1)^{-1}a_3^1\\
b_2^2 &= b_4^1 - a_4^1(b_3^1)^{-1}c_3^1 - c_4^1(b_5^1)^{-1}a_5^1\\
b_3^2 &= b_6^1 - a_6^1(b_5^1)^{-1}c_5^1 - c_6^1(b_7^1)^{-1}a_7^1\\
a_2^2 &= -a_4^1(b_3^1)^{-1}a_3^1\\
a_3^2 &= -a_6^1(b_5^1)^{-1}a_5^1\\
c_1^2 &= -c_2^1(b_3^1)^{-1}c_3^1\\
c_2^2 &= -c_4^1(b_5^1)^{-1}c_5^1\\
d_1^2 &= d_2^1 - a_2^1(b_1^1)^{-1}d_1^1 - c_2^1(b_3^1)^{-1}d_3^1\\
d_2^2 &= d_4^1 - a_4^1(b_3^1)^{-1}d_3^1 - c_4^1(b_5^1)^{-1}d_5^1\\
d_3^2 &= d_6^1 - a_6^1(b_5^1)^{-1}d_5^1 - c_6^1(b_7^1)^{-1}d_7^1.
\end{aligned} \tag{4}$$

The system in Eqs. (3) is of the same form as the original problem but has the lower rank $= 3$. This suggests, by induction, that one can successively reduce the system until its rank is one. The solution of such a simple system then leads to all the other solutions by way of the substitution formulae such as those given in Eqs. (2). The several terms in the Eqs. (4) that have three matrix factors will be called the triplet terms. Since they are all of the same form one method of evaluation will extend to all of them. Before describing the details of the remaining levels of reduction, let us consider how Eqs. (4) are parallelized.

First, an inverse representation of the indicated blocks must be obtained. That is, $(b_1^1)^{-1}$, $(b_3^1)^{-1}$, $(b_5^1)^{-1}$ and $(b_7^1)^{-1}$ must be represented. The PAMS code allows one to choose either the actual matrix inverse or an lu factorization. We allow this choice because it is not clear which will ultimately run faster. Linear algebra tells us the factorization-backsolve approach always has fewer arithmetic operations than the inversion-multiplication alternative. However, this latter method is more highly optimized in existing software and runs faster. We keep the former because it may be better optimzed in the near future.

In the above formulae of Eqs. (4) we note the inverse operates on a block matrix or on a sub-vector. Depending on the choice of the representation of the inverse, we either perform matrix multiplication or we apply backsolves to obtain the result. For example, the first block matrix problem to be encountered in evaluating Eqs. (4) is, say,

$$h = (b_1^1)^{-1}c_1^1.$$

In the one version the actual inverse matrix is used to multiply c_1^1 to obtain h. For the other version we solve the equivalent problem

$$b_1^1 h = c_1^1$$

by first generating lu factors for b_1^1 which are then used in the familiar backsolve operations to obtain h. When h is a matrix, as in this specific example, there are really j such systems to solve where j is the rank of the block.

The remaining operations in Eqs. (4) are trivial matrix multiplications and additions which are easily implemented.

Having now reviewed some of the details about how we carry out the operations on the blocks, we turn to examine Eqs. (4) to learn what procedures are independent so that multitasking may be applied.[9–12] As they stand, the formulae in Eqs. (4) are not independent because the b inverses couple many of them together. But by generating these formulae in two stages, we do find sufficient independence to allow multitasking. At the first stage we compute

$$\begin{array}{lll} (b_1^1)^{-1}c_1^1 & (b_1^1)^{-1}d_1^1 & \\ (b_3^1)^{-1}a_3^1 & (b_3^1)^{-1}c_3^1 & (b_3^1)^{-1}d_3^1 \\ (b_5^1)^{-1}a_5^1 & (b_5^1)^{-1}c_5^1 & (b_5^1)^{-1}d_5^1 \\ (b_7^1)^{-1}a_7^1 & (b_7^1)^{-1}d_7^1. & \end{array} \tag{5}$$

We have displayed on separate lines those procedures that are independent. Thus the indicated operations can be done in four tasks to be run in parallel on separate processors.

In the second stage of the evaluation of Eqs. (4) we compute

$$\begin{array}{llll} b_1^2 & c_1^2 & d_1^2 & \\ a_2^2 & b_2^2 & c_2^2 & d_2^2 \\ a_3^2 & b_3^2 & d_3^2 & \end{array} \tag{6}$$

in three independent tasks. None of the operations in the second stage may commence until all of the results of the first stage are completed.

Now we proceed to the second level of reduction in which $k = 2$ and $i = 1, 2$. We thus solve the odd rows from Eq. (3) to get

$$\begin{aligned} x_2 &= (b_1^2)^{-1}(d_1^2 - c_1^2 x_4) \\ x_6 &= (b_3^2)^{-1}(d_3^2 - a_3^2 x_4). \end{aligned} \tag{7}$$

By substituting these into the even row(s) of Eq. (3) we obtain a single equation for x_4:

$$b_1^3 x_4 = d_1^3.$$

Here, the coefficients at the next level are,

$$\begin{aligned} b_1^3 &= b_2^2 - a_2^2(b_1^2)^{-1}c_1^2 - c_2^2(b_3^2)^{-1}a_3^2 \\ d_1^3 &= d_2^2 - a_2^2(b_1^2)^{-1}d_1^2 - c_2^2(b_3^2)^{-1}d_3^2. \end{aligned} \tag{8}$$

The operations in (8) are treated in two stages just as was done for the Eqs. (4). There is less oppourtunity for multitasking here because one can see the first stage only admits two independent tasks and the second stage just one.

At this point we have completed the reduction phase of the calculation by having reduced our original rank $N = 7$ system of blocks down to a single block equation. We solve this equation to obtain x_4 either by inversion or by factorization. Clearly, this step cannot be multitasked.

What we call the synthesis phase of the calculations begins here. Using the several formulas already developed we rapidly generate all of the other sub-vectors from x_4. This is done by next evaluating Eqs. (7) to find x_2 and x_6 by using the inverse representation already computed (and saved) for the operators $(b_1^2)^{-1}$ and $(b_3^2)^{-1}$. The two tasks found here allow some parallel operation via multitasking.

Lastly, we perform the operations of Eqs. (2) to obtain the remaining unknown sub-vectors x_1, x_3, x_5, and x_7. Four tasks exist here which allow full multitasking on the four processor Cray-2.

With regard to what can be multitasked, we have identified two stages at every level of reduction and one stage at every level of synthesis. If one does an operation count, one finds almost all of the work is done in the reduction phase. In the example we give here, only one level could be fully multitasked. As is shown by the more general algorithm, given in the appendix, more practical problems will have a larger system of blocks which implies more levels of reduction. All of these levels will fully multitask except for the last two of which one partially multitasks. At each level the work to be done is just proportional to the number of rows of blocks. From these considerations we derive a formula for the maximum multitasking overlap that can be achieved:

$$\mathcal{M} = \frac{4N}{[illegible] + N}$$

Figure 2 shows the maximum possible multitasking overlap as a function of the number of block rows. For 64 block rows, which may be regarded as typical, one could achieve an overlap of $\mathcal{M} = 3.7$. This confirms an earlier suspicion that multitasking cyclic reduction is a viable technique. Later, after we present some related problems and their solution strategies, we shall present results from the Cray-2 of runs of PAMS on the matrix A_1.

As presented the PAMS method appears to require the storage of $2N$ blocks for the types a, b, and c. First, we point out that should we wish to solve for more than one right hand side then we must store all of the a and c blocks since the even numbered ones must be saved (at each level)

PAMS Storage in Numbers of Blocks		
	Unitasking	Multitasking
Single Solution	$3N$	$4N$
Multiple Solutions	$5N$	$5N$

Table 1: The requirments for storing the blocks depend on which arrays may be overwritten.

inorder to treat subsequent Y vectors. If just one problem is to be solved, then we can overwrite some of the blocks: namely, we can overwrite the even numbered blocks a, b, and c as we proceed through the reduction levels. Were this a unitasking algorithm, only the blocks appearing in the elimination formulae (e.g. Eqs. 2 and 7) need be saved. Thus for the single problem in unitasking mode this suggests the use of a more compact storage scheme where the block matrices appearing at the higher levels would find storage in the $k = 1$ level by overwriting the even numbered blocks. Thus one need only store N blocks each for the a, b, and c types when unitasking. For the circumstance of multitasking, additional constraints arise which complicate this technique and lead to the requirement of an extra N blocks of scratch storage for the first reduction level. The implication is that multitasking increases the storage requirement from at least $3N$ blocks to at least $4N$ blocks for the problem A_1. Where one has multiple problems, the storage requirement goes to $5N$ both for unitasking and multitasking. In the Table 1 we show PAMS storage requirements for the blocks. Each block has j^2 words where j is the block rank. To keep the PAMS code most general we have chosen to not overwrite the even numbered a and c blocks, but it will be straightforward to reduce the storage for cases in which it is warranted.

3. Related Sparse Systems from Plasma Physics

3.1. *The Matrix Form A_2*

Another matrix problem that is encountered in plasma physics equilibrium theory and other disciplines is also block tridiagonal except the blocks are not dense. In 3D representations where finite elements (or differences) are used in two or more of the coordinates one can specify the linear system with a matrix that contains blocks that are banded and often sparse. For 2D problems in which both coordinates are represented by finite elements a matrix like this arises also.[3] In Fig. 3 we display the artist's view of such a matrix which we denote by A_2.

The method of solution is very similar to what we described above. We could use the method for the matrix A_1 but it would not take advantage of the banded structure of the blocks and thus would be wasteful of computer memory and time. At the first stage we also eliminate half of the sub-vectors by the same formulation given in Eqs. (2). And we also get the same expressions for the reduced system (Eqs. (3)) and forumulas for the new coefficient blocks(Eqs. (4).) In the method we gave for A_1 we permitted the option of using inverses or factorized representations. For A_2 we restrict the method to using factorization so that one can use band solvers or sparse matrix techniques to implement the formulas of Eqs. (2) and of Eqs. (4). At this first level all the matrix blocks are stored in compressed form typical in band solvers. If the matrix blocks are also sparse one can employ storage schemes and solvers that are even more economical than those of the band solvers. The sparse or banded structure does not carry over to the subsequent levels of reduction. In fact the blocks formed on the left hand side of Eqs. (4) are dense. Thus for the second level of reduction, and all higher levels, one uses the same method used for A_1. In the synthesis phase of obtaining all the x sub-vectors, one proceeds as in A_1 until one reaches the last level. At the last level the appropriate sparse or banded version of Eqs. (2) are used to generate the remaining sub-vectors x.

Approximately half of the storage requirement and half of the arithmetic operations are used in the first level of the algorithm given for A_1. In the algorithm for A_2 the banded or sparse structure at the first level requires comparatively little storage and much less arithmetic. This suggests that the method for A_2 will save up to a factor of 2 in both memory and computer time as compared to using the full PAMS method. Further, we have learned that once the first level of reduction is done one then uses the algorithms of PAMS to complete the solution of the reduced system obtained. From a coding viewpoint, one would retain PAMS and add routines to carry out the first level of reduction and the last level of synthesis. In some sense matrix A_2 is a special case of A_3 to be described next. Some details of the implementation of the first level treatment of matrices A_2 and A_3 are given in Appendix B.

3.2. The Matrix Form A_3

The problem for A_3 is shown pictorially in Fig. 4. The structure of A_3 is not strictly block tridiagonal and close inspection of the Figure shows that there are four types of blocks. There are small and large square blocks and there are other rectangular blocks with either a vertical or horizontal orientation.

One type of plasma calculation that produces this form of matrix is that of the linear stability of two dimensional equilibria when hybrid finite elements are used.[4,1] In this approach a different set of basis functions is used to expand the radial derivatives than would be obtained by simple differentiation of the radial basis functions. It is found that this approach is very accurate in spite of its use of so-called non-conforming elements; it largely eliminates the pathological condition known as spectral pollution as described in the cited references.

In the 2D case there are three spatial components to the displacement field. If one orders the discrete equations properly the matrix structure A_3 is obtained. The x vectors contain the radial displacements while the y vector contains both the azimuthal and axial displacements. In the finite element expansion there are equal numbers of basis functions for radial, axial, and azimuthal displacement expansions. Since the y vector combines the latter two together, it will have twice the number of unknowns as the x vector.

In our development of the 3D linear stability code we are also intending to use hybrid elements in the radial coordinate. A matrix very similar to the one we call A_3 results. Obviously, there are many variations of the detailed structure of A_3 for the 3D stability calculation depending on the choice of basis functions. Nevertheless, many of them will share a form like A_3 that will enable one to use very similar techniques. As in the foregoing example for the solution of the matrix A_1 we find it useful to write out an example.

The system to be solved is written

$$\begin{aligned}
b_1^1x_1 + c_1^1y_1 + w_1^1x_2 &= d_1^1 \\
a_2^1x_1 + b_2^1y_1 + c_2^1x_2 &= e_1^1 \\
u_3^1x_1 + a_3^1y_1 + b_3^1x_2 + c_3^1y_2 + w_3^1x_3 &= d_2^1 \\
a_4^1x_2 + b_4^1y_2 + c_4^1x_3 &= e_2^1 \\
u_5^1x_2 + a_5^1y_2 + b_5^1x_3 + c_5^1y_3 + w_5^1x_4 &= d_3^1 \\
a_6^1x_3 + b_6^1y_3 + c_6^1x_4 &= e_3^1 \\
u_7^1x_3 + a_7^1y_3 + b_7^1x_4 &= d_4^1
\end{aligned} \tag{9}$$

In all of the blocks of Eqs. (8) we have sparse matrices. To solve this system we eliminate all the unknown sub-vectors y_i for all i. Doing this we have,

$$\begin{aligned}
y_1 &= (b_2^1)^{-1}(d_1^1 - a_2^1x_1 - c_2^1x_1) \\
y_2 &= (b_4^1)^{-1}(d_2^1 - a_4^1x_2 - c_4^1x_3) \\
y_3 &= (b_6^1)^{-1}(d_3^1 - a_6^1x_3 - c_6^1x_4)
\end{aligned} \tag{10}$$

From the substitutions we make the following identifications:

$$\begin{aligned}
b_1^2 &= b_1^1 - c_1^1(b_2^1)^{-1}a_2^1 \\
b_2^2 &= b_3^1 - a_3^1(b_2^1)^{-1}c_2^1 - c_3^1(b_4^1)^{-1}a_4^1 \\
b_3^2 &= b_5^1 - a_5^1(b_4^1)^{-1}c_4^1 - c_5^1(b_6^1)^{-1}a_6^1 \\
b_4^2 &= b_7^1 - a_7^1(b_6^1)^{-1}c_6^1 \\
a_2^2 &= u_3^1 - a_3^1(b_2^1)^{-1}a_2^1 \\
a_3^2 &= u_5^1 - a_5^1(b_4^1)^{-1}a_4^1 \\
a_4^2 &= u_7^1 - a_7^1(b_6^1)^{-1}a_6^1 \\
c_1^2 &= w_1^1 - c_1^1(b_2^1)^{-1}c_2^1 \\
c_2^2 &= w_3^1 - c_3^1(b_4^1)^{-1}c_4^1 \\
c_3^2 &= w_5^1 - c_5^1(b_6^1)^{-1}c_6^1 \\
d_1^2 &= d_1^1 - c_1^1(b_2^1)^{-1}e_1^1 \\
d_2^2 &= d_2^1 - a_3^1(b_2^1)^{-1}e_1^1 - c_3^1(b_4^1)^{-1}e_2^1 \\
d_3^2 &= d_3^1 - a_5^1(b_4^1)^{-1}e_2^1 - c_5^1(b_6^1)^{-1}e_3^1 \\
d_4^2 &= d_4^1 - a_7^1(b_6^1)^{-1}e_3^1
\end{aligned} \tag{11}$$

All of the operations in Eqs. (10) and (11) are sparse matrix operations which lead to full blocks a_i^2, b_i^2, c_i^2 at the second level. As in the two preceding matrix problems, the independent structures of Eqs. (10) and (11) are analagous to those we described for Eqs. (2) and (4). The multitasking of the two stages is done just as before. What differs here are the manipulations within the blocks. At the first level some of the blocks are not square. Only the square ones need an inverse representation which will be done by factorization only. The rectangular ones only come in as multiplier or backsolve operands so the treatment is straightforward. In appendix B we present some details of the sparse treatment of the first reduction level for the solution of the problem A_3 (of which A_2 is a special case.) Problems A_3 and A_2 share the feature that once the first level of reduction is completed, one proceeds to solve the reduced problem by the method in PAMS.

In this and the preceeding sections we have presented various block matrix problems that arise in some physics calculations in three dimensions. All of these problems reduce to the same matrix structure after an initial reduction procedure is completed. At that point all of them can be solved by the methods employed in the PAMS code.

4. The Running of PAMS

The PAMS code attempts to exploit several types of parallel operation at once. We have shown, by way of an example, what procedures are independent so that they can be multitasked. Within the tasks we use the various basic linear algebra subroutines (BLAS) which we now discuss from the point of view of obtaining the other forms of parallelization.

Aside from trivial matrix (or vector) additions and subtractions we use the Cray Research Incorporated (CRI) routines[5] MINV, MXM, and MXV as well as the LINPACK routines[6] SGEFA and SGESL. They invert a block, form a block times block product, form a block times sub-vector product, factorize a block, and backsolve a block on a sub-vector, respectively. These routines are intended to be highly optimized by exploiting vectorization, by overlapped operation of the two floating point units (one each for addition and multiplication), by overlapping memory traffic with the above, and by use of the fast local memory for the storage of intermediate results. The parallelization of vectorization is easily introduced by properly written fortran code. But to do a good job of obtaining what we shall call the parallelization of overlap it is often necessary to write the routines in assembly language (Cal-2 in this case.)

We have identified three types of parallelization, namely: vectorization, functional unit overlap, and multitasking. In the Cray-2 environment, the first of these permits one to generate a floating point result every clock period (4.1 nanoseconds). Since there are just two functional units in each CPU, perfect overlap of them would allow another factor of two beyond vectorization so that a result would be generated every 2.05 nanoseconds. We are assuming that we have also overlapped all memory traffic so that it never interferes with the arithmetic processing. Combining this performance with full multitasking gives another factor of 4 improvement. Thus the theoretical performance of the Cray-2 could be as fast as one result every .5125 nanoseconds. Expressed as floating point results per unit time we get 1.951 gigaflops (1951 megaflops.)

In this paper we are not solving a specific physical problem, but are demonstrating the PAMS technique for a class of problems that lead to matrices of the given forms. In this section we present the results of test runs in which we formulated the matrix operators from random numbers and subsequently imposed certain asymmetries and diagonal dominance properties. Both our interest in demanding 3D physics problems and our curiosity about the Cray-2 performance on very large memory problems led to our choice of the problem we present here. We decided on a problem with 64 block rows; this corresponds to 64 radial zones of a finite element (or finite difference) method. In the other angular coordinates we decided to use a total of 319 global basis functions. It would be unlikely that a real application would choose 319 because it must be the product of the dimensions of the two 1D subspaces; here only 11 and 29 are acceptable 1D dimesnions. Nevertheless, we chose this number because it was very close to a threshold where the implied code size begins to exceed the memory limit of 40 million words that is currently enforced at Livermore. We chose the specific number 319 because it is just one less than a multiple of 64. This means the vectorization will be very close to optimal (a mulitple of 64 would be slightly better if vectorization were the only issue.) The choice of an odd number just less than a multiple of 64 gives optimal performance of the memory fetches and stores because it minimizes bank and quadrant conflicts. A slightly more general MXM would allow multiples of 64 and at the same time suppress bank conflicts by using a different storage dimension on the leading index than the matrix dimension; this storage dimension would typically be one larger than the multiple of 64. MINV already allows the distinction between matrix dimension and storage dimension. By these choices we felt that vectors would be long and tasks would be big so that the parallel performance might approach its theoretical asympotitic values.

Once given a problem, the bulk of all the calculation is done in three subroutines PAMSPREP, PAMSTASK, and PAMSWORK. During the upward sequence of reduction the first two of these are called at each level.

PAMSPREP requires the generation of the inverse representation either by calling MINV (to get the actual inverse) or by calling SGEFA (to obtain the *lu* factors.) Then either MXM or SGESL is used with the inverse representation to generate the right-hand pair of the triplets. At the time of the writing of this article, the routines MXM and MINV have been very well optimized for the Cray-2 by CRI. The bulk of these routines are written in Cal-2 assembly language to insure vectorization, good functional unit overlap, and exploitation of the local memory. The alternative routines, SGEFA and SGESL (of LINPACK), are written in fortran and as of yet do not have fast assembly language versions; thus we cannot presently obtain the desired speeds from their use. By using MINV and MXM we estimate unitasking speeds of about 340 Mflops for PAMSPREP from which we can infer gigaflop speeds when it is multitasked.

PAMSTASK has the job of computing the triplets and then adding them together to form the matrix blocks (and sub-vectors) at the next reduction level. MXM and MXV perform most of these operations. Almost all of the work is done by MXM thus PAMSTASK is expected to be slightly faster than PAMSPREP and somewhat above a gigaflop when multitasked.

Lastly, PAMSWORK carries out the generation of the x's from the elimination formula and uses MXV or SGESL depending on the inverse representation chosen. MXV is adequately optimised thus suggesting that PAMSWORK will run well. Counting operations shows that PAMSWORK does very little compared to the preceeding two subroutines so its performance is not crucial to the overall speed.

Before displaying some typical results from PAMS we list and explain several of the headings seen in the output:

jdim The rank or order of the block matrices.

ndim The number of block rows in the global matrix.

np The number of reduction levels.

nmtskon For `nmtskon` = 0 (1) the code unitasks (multitasks).

inmon For `inmon` = 0 (1) the code uses LU factorization (uses the inverse).

iforon For `iforon` = 0 (1) the code uses Cal-2 assembly language if available (uses fortran).

pamstask The designation **`pamstask:` k=n** denotes that subsequent output gives the performance for the subroutine **`pamstask`** at the reduction level $k = n$.

upred The designation **upred: k=n** denotes that subsequent output gives the performance of the entire reduction level $k = n$ which includes the subroutines **`pamstask`** and **pamsprep**.

dynsyn This designation precedes the output of the synthesis performance statistics for the work performed in the subroutine **`pamswork`**.

global statistics The output following this title gives the performance statistics for the entire solution process.

CAL inv/mxv This line indicates the assembly language Cal-2 was used where needed and that matrix inversion was chosen.

relative difference The error of the X vector is given as the relative $L2$ norm.

With these terms defined we present here the output from our best run to date:

```
      jdim       ndim          np       nmtskon     inmon     iforon

       319        64            6          1           1         0


  pamstask: k= 2
       CPU sec.      Sys sec.        CPU overlap    Gigaflops
       23.962390      .005633        3.690229       1.266691
 upred:   k=2
       CPU sec.      Sys sec.        CPU overlap    Gigaflops
       44.485919      .027430        3.751080       1.220194
  pamstask: k= 3
       CPU sec.      Sys sec.        CPU overlap    Gigaflops
      11.787837        .005048        3.738749       1.283692
upred:   k=3
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
      21.981358        .010818        3.656341       1.192673
 pamstask: k= 4
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
        5.585660       .005085        3.182449       1.115797
upred:   k=4
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
      10.510431        .028172        3.498240       1.171514
 pamstask: k= 5
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
        2.585590       .005685        3.088537       1.091687
upred:   k=5
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
        5.005815       .012302        2.818901        .954285
 pamstask: k= 6
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
         .979820       .002080        1.199246        .479391
upred:   k=6
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
        2.068654       .004769        1.236935        .467610
 pamstask: k= 7
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
         .325788       .000900        1.001476        .401340
upred:   k=7
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
         .850528       .002251        1.003600        .384360
upred:   k=8
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
         .207725       .000606         .997824        .311865
dnsyn:   k=7
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
         .005726       .000613         .965110        .051401
dnsyn:   k=6
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
         .006691       .000612         .970554        .088472
dnsyn:   k=5
       CPU sec.       Sys sec.       CPU overlap    Gigaflops
```

```
        .012838       .001651        .984660       .109147
dnsyn:  k=4
       CPU sec.      Sys sec.       CPU overlap   Gigaflops
        .025932       .000606        .992233       .116676
dnsyn:  k=3
       CPU sec.      Sys sec.       CPU overlap   Gigaflops
        .050567       .000605        .996021       .124128
dnsyn:  k=2
       CPU sec.      Sys sec.       CPU overlap   Gigaflops
        .099107       .000617        .997946       .128957
dnsyn:  k=1
       CPU sec.      Sys sec.       CPU overlap   Gigaflops
        .198693     1 .002554       1.002867       .130307

global statistics

       CPU sec.      Sys sec.       CPU overlap   Gigaflops
      85.524900     2 .106610       3.315195      1.088404



CAL  inv/mxv

relative x difference =  0.35366e-13
```

Multitasking is giving close to optimal overlaps even when measured over the entire run. Vectorization seems to be running near its asymptotic speed. We see a sustained speed of 1.267 gigaflops during the first reduction phase in the subroutine PAMSTASK (for 27 seconds of CPU time.) Averaged over the entire code run we measure the speed to be 1.088 gigaflops.

The Cray-2 installation at Livermore uses the very interactive multitasking time sharing system CTSS. In this dynamic environment, the operating system scheduler cannot start multiple tasks simultaneously but must start the various tasks as processors become available. Typically, in this environment, multiple tasks start within several milliseconds of each other. This sequence is dependent on the nature of the competing codes that are sharing the memory and thus the exact amount of multitasking overlap will depend on these statistical details. In several runs of the PAMS code we have seen roughly 10% variations in the speeds reported which seems consistent with the CTSS system. The results displayed here were obtained from a special "stand alone" run in which time-sharing was suppressed. When run under time-sharing mode, the speeds are still quite high- about 5% slower than the above results.

The PAMS code together with its driver routine sets up matrix problems with known solutions. It does this by the artifice of choosing the solution first and then generating the right-hand side by matrix multiplication. The so-called true solution is saved while the PAMS routines carry out the algorithm. The obtained solution is compared with the saved one. The solution is checked further by multiplying it by the matrix to obtain a right-hand side vector to be compared with the given source vector. In the results presented above, the errors shown are consistent with the roundoff errors in the Cray-2. We show the results of the CPU time, the system time, the overlap, and the gigaflops at each of the synthesis and reduction levels and additionally give these numbers for the subroutine PAMSTASK which was so well optimised. The performance of PAMSPREP is almost as good. We have learned of other optimal routines that may be used to replace MINV and MXM (if we measure them to be faster.) Thus we may be able to get about 10% further speedup beyond what is reported here.

5. Summary and Conclusions

Led by the desire to make more realistic 3D calculations of toroidal plasma equilibria and their linear stability properties, we have made an effort to exploit the features of the best scientific computer now available. The very large memory, of the Cray-2, gives it a capability to compute on a reasonably refined grid without needing to use disk memory. Its large memory also allows us to employ certain direct matrix solvers that would be forbidden on the smaller machines often used. We have shown, by way of example, that multitasking works extremely well and that implicit problems, such as those presented here, can be treated with methods that exhibit large independent processes amenable to this form of parallelism. In our efforts to get good performance we have tried to use the other two forms of parallelism of vectorization and of functional unit overlap. We have control over vectorization in the fortran environment, but not over the functional units- given the compilers in use. Thus we have had to rely on the best assembly language routines available to implement the basic manipulations within the blocks. We have seen gigaflops in those code segments that use well optimized linear algebra routines. Fortunately, in this application, almost all of the linear algebra was of this requisite form. This suggests that other applications, many of which are not as implicit as this one, should also be amenable to solution by comparably fast highly optimsed algorithms. The Cray-2, indeed, may be seen as the machine it first promised to be- unequaled in its power and capability among its contemporaries.

5.1. Limitations of the PAMS Method

We must address the question as to when PAMS is inferior to other methods? The answer depends on the criterion measuring its merit. The cost, the operation count, the speed, the memory requirments, and the ease of implementation are all relevant.

First, let us consider the unitasking environment. After many years of development and experience, direct band solvers are often found to be the method of choice. Their nearest competitors are usually the iterative methods such as preconditioned conjugate gradient algorithms which are prescribed when the rank of the global matrix gets too large. Should the direct solver be chosen, one proceeds by using an *LU* factorization of the global banded matrix afterwhich backward and forward solves are done to obtain the solution. If the half-bandwidth is κ (it does not count the diagonal) then a rank n matrix requires about $2n\kappa^2$ floating point arithmetic operations for its solution. Alternatively, counting the operations in the cyclic reduction method we find it requires a factor of 1.75 more operations. Both methods afford roughly equal levels of optimization through vectorization and functional unit overlap. Thus the cyclic reduction technique is not viable in a unitasking computer.

In a multitasking computer, with p CPU's, the multiple processors can take advanatge of the favorable structure of the cyclic reduction algorithm to give overall performance gains on the order of $p/2$ compared to the unitasked band solver. As shown above, multitasking requires more storage (about a third more) for single problems having one right hand side. For multiple right hand sides the storage remains the same as in the unitasking case. In future more massively parallel computer architectures considerably higher performance gains are inferred.

So far we have used performance as the sole criterion. If we assume that computer costs are charged on a per cycle basis, then the PAMS method is always more costly except in the very large memory limit. For using nearly all of the memory it is assumed that a unitasking PAMS would block access to the other processors and would consequently be charged for the engendered idle cycles. Unfortunately, costs are not charged in such a simple fashion. In the MFECC Cray-2 installation, a large charge is made for the use of memory only during those intervals when the code is running. Multitasking can reduce this memory rent considerably up to a factor of four.At the current user memory limit of 40 million words the memory charge is 3.6 times the CPU charge per unit time unitasking. In fully overlapped multitasking it is only .6 times as much. In the two cases then the total charges are 4.6 and 1.6 per unit CPU time respectively. This gives multitasking a cost advantage by a factor of about 2.9 per unit CPU time (or per cycle of computation.) Folding in the extra cycles required for the PAMS method we arrive at a relative cost advantage of 1.7 for using PAMS.

Another possible algorithm uses Gaussian elimination on the blocks such that the first task eliminates the unknowns from the top down while the second task eliminates them from the bottom up until a reduced system with two block rows is obtained. Such an algorithm has the same amount of arithmetic as the band solver but is limited to only two tasks so is not attractive for massively parallel computers. It might be suitable for the two processor Cray X-MP 22 at Livermore but for the four processor Cray-2 it is questionable.

In summary, we estimate the fully optimized PAMS will be superior in performance to a band solver and in the MFECC environment more economical as well. This latter advantage is critically dependent on the charging policy of the computer center; changes in the charging algorithm could quite possibly put PAMS in an inferior position realtive to the cost of a band solver.

We feel that the combination of the more capable Cray-2 computer with new numerical techniques such as described here will allow us to routinely calculate physical phenomena that was heretofore impractical and slow. The PAMS method provides another example that implicit systems (such as those that yield block-banded matrices) can be multitasked with good multitasking overlap. Possible drawbacks to the method are its somewhat larger memory requirement and its greater number of arithmetic operations which could conceivably translate into higher costs.

Appendix A. PAMS Formulae and Data Structure

All of the linear systems in the PAMS analysis are of the form

$$a_i^k x_{(i-1)2^{k-1}} + b_i^k x_{i2^{k-1}} + c_i^k x_{(i+1)2^{k-1}} = d_i^k \tag{A1}$$

where at each level k the index i ranges over

$$i = 1, \ldots, n(k). \tag{A2}$$

The given first system of equations has $k = 1$ and $n(k) = N$. At the subsequent levels of reduction we have

$$n(k) = \frac{n(k-1)+1}{2} \qquad 1 < k \le K \tag{A3}$$

where $K = 2 + \log_2(N-1)$. It is to be understood that the integer formulae always truncate to the integer value not greater.

The recursion formulae for the block matrices, that come from the substitution step, are:

$$\begin{aligned} a_i^k &= -a_{2i}^{k-1}(b_{2i-1}^{k-1})^{-1}a_{2i-1}^{k-1} \\ b_i^k &= b_{2i}^{k-1} - a_{2i}^{k-1}(b_{2i-1}^{k-1})^{-1}c_{2i-1}^{k-1} - c_{2i}^{k-1}(b_{2i+1}^{k-1})^{-1}a_{2i+1}^{k-1} \\ c_i^k &= -c_{2i}^{k-1}(b_{2i+1}^{k-1})^{-1}c_{2i+1}^{k-1} \\ d_i^k &= d_{2i}^{k-1} - a_{2i}^{k-1}(b_{2i-1}^{k-1})^{-1}d_{2i-1}^{k-1} - c_{2i}^{k-1}(b_{2i+1}^{k-1})^{-1}d_{2i+1}^{k-1} \end{aligned} \tag{A4}$$

for $i = 1, \ldots, n(k)$ at each level k. These blocks are computed recursively in an ascending sequence of k's for $k = 2, \ldots, K$. At the top most level we have a special case of the preceding formulae, namely:

$$\begin{aligned} b_1^K &= b_2^{K-1} - a_2^{K-1}(b_1^{K-1})^{-1}c_1^{K-1} \\ d_1^K &= d_2^{K-1} - a_2^{K-1}(b_1^{K-1})^{-1}d_1^{K-1}. \end{aligned} \tag{A5}$$

The rank one system obtained at the top most level gives the special formula for computing the Nth sub-vector x_N:

$$x_N = (b_1^K)^{-1} d_1^K. \tag{A6}$$

Next, we recall the elimination formulae that were used in the derivations of the reduced systems. For this we have,

$$x_{i2^{k-1}} = (b_i^k)^{-1}[d_i^k - a_i^k x_{(i-1)2^{k-1}} - c_i^k x_{(i+1)2^{k-1}}] \tag{A7}$$

which is evaluated for $i = 1, \ldots, n(k)$ at each level k. These are performed in a descending sequence of k's:

$$k = K, \ldots, 1.$$

The formulae given here are generalizations of those given in Section 2. Here we have presented the formulae in the order they are computed whereas earlier we gave them in the order they were derived.

To describe the multitasking[9–12] of these procedures, we first recall that substitution formulaes are evaluated in two stages within each level k. When this is done, all of the operations denoted by the subscript i are independent of all the other i's. The simplest approach would be to generate $n(k)$ separate tasks at each level k. We believe that it is most efficient to limit the number of tasks to equal the number of processors because this minimizes the multitasking overhead and in this particular calculation tends to preserve the load balancing (which means keeping the amount of calculation the same per task.) In our coding we have replaced the unitasking construct

```
      do loop i = 1, nok
      ...
      ...
 loop continue
```

with the structure

```
      do outer jk = 1, 4
      call swork(jk)
 outer continue
```

where the subroutine SWORK contains a partitioned loop as follows:

```
      subroutine swork(jk)
      ...
      do inner i = jk, nok, 4
      ...
      ...
 inner continue
      return
      end
```

In this form, there are four completely independent procedures each identified by the value of `jk`. As given here multitasking would not occur unless we also include the appropriate multitasking commands. To invoke the multitasking of this example we replace the loop "`do outer`" with

```
      do outer jk = 1, 4
      call tskstart(idtsk(1,jk),swork,jk)
 outer continue
c..Provide barrier synchronization
      do barr jk = 1, 4
      call tskwait(idtsk(1,jk))
 barr continue
```

In this example, the variable `idtsk` is a task identifier array used by the system to keep track of the several tasks. The call to `tskstart` has the effect of starting the subroutine `swork`, but it does not wait for the return. The loop `outer` is rapidly completed at which time all of the tasks have been started but not finished. To provide simple barrier synchronization the second loop `barr` employs the call to `tskwait` which has the effect of stopping the calling code until each and every task has finished.

The storage of the matrix variables is done as follows. The blocks a_i^k and c_i^k and the sub-vectors d_i^k are stored in arrays that have just more than twice the nominal storage needed for just level $k = 1$. In the case of the blocks a_i^k the storage declaration is

```
dimension a(jdim,jdim,2*ndim+1)
```

In this example, `jdim` is the rank of the blocks and `ndim` is the rank of the global matrix in terms of blocks- its block rank. The second level blocks are stored in the third quarter of this array, those of the third level occupy the eighth just beyond that, and subsequent blocks squeeze in the remaining space. In the derivations shown in this paper one might infer that b_i^k and its inverse are stored in such an extended array. However, it is possible to mix the b inverses at the various levels k into a smaller array dimensioned

```
dimension bi(jdim,jdim,ndim).
```

A similar declaration is used for the b's as well. In many applications, the b's are not needed later in which case it is permissible to overwrite them with the inverse representations.

Appendix B. Details of A_3's First Level Reduction

Most of what follows is also true for matrix problem A_2 which is a special case of A_3.

In problem A_3 there are two different block dimensions and it is ambiguous as to the block count. Do we count the superblocks that overlap, or do we count the sum of the little blocks plus the big blocks? It does not matter as long as we are consistent with our starting definition. The notation we have adopted suggests we count all the block rows; in our example there are seven block rows at the beginning. Another property to note is that it was straighforward to eliminate the y's in terms of the x's. The cyclic reduction procedure will not permit elimination of the x's in terms of the y's because the couplings are more complicated and involve other sub-vectors than those that could be characterized as nearest neighbor. Since the blocks at the first level are sparse we must employ this knowledge to greatly reduce the amount of arithmetic to be performed. We proceed in two stages as we did above. In the first stage the calculation of

$$h = (b_2^1)^{-1} a_2^1$$

will suffice to give the method for all of the products of the right-hand pair of factors in the triplets in Eqs. (11)

Let the rank of the block (the number of rows in a block) be j. Whether banded or sparse, the matrices b_2^1 and a_2^1 have many fewer non-zero elements in them than if they were dense. In this context we shall regard an algorithm as sparse if it computes just on the non-zero elements. When b is non-symmetric we compute its lu factorization. If it is symmetric then we obtain its Cholesky factorization ldl^t. Let us consider the case where

$$lu = b$$

We wish to obtain h from

$$luh = a.$$

Let $g = uh$ so that we are solving

$$lg = a$$

for g. In obtaining the factors l and u which are respectively lower and upper triangular, the phenomenon of fill-in occurs but the l and u matrices remain banded within the same band boundaries possessed by b. Since g and a are matrices we can solve the above equation by partitioning each of these into their several column vectors which are then solved by the Gaussian elimination technique. The backsolve algorithm for obtaining g from the previous equation begins in the first row where the l matrix has only one element. It then proceeds to the subsequent rows and in one sweep through the rows generates all of the g elements in a given column of g. Once all the columns are processed it can be shown that the matrix g is lower triangular and dense in that region. Next, the equation $uh = g$ is treated by applying backsolves to the column vectors within h and g; this procedure starts in the last row and proceeds upward generating each column vector of h in one pass. h is a full dense matrix. Since u is sparse or at least banded the number of operations is reduced, even with g being lower triangular dense. Performing the next product in the triplet is also done quickly because the matrix of the left factor is sparse or banded.

The operations we have described here can be performed quickly because in each case the number of arithmetic operations scales as j^2. If we had begun with dense matrices the scaling would have been j^3.

The other possibility that may be favored would be to apply cyclic reduction on these block problems and use the independent procedures as a basis for vectorization. This could be done if the blocks possessed a sub-block form. We suggest that these options must be tested to see which alternative is preferable for a given class of problems. The choice to make will depend on the computer hardware characteristics, the relative optimization of the algorithms, and to some extent the ease of programming it.

In this description of some of the details of carrying out the block manipulations in the first reduction stage of problem A_3 we have not mentioned multitasking. The algorithm for multitasking problem A_3 is precisely the same as for A_1 and A_2 so it is not necessary to repeat. The only differences are in the techniques used for operations within the blocks which do not affect the multitasking algorithms.

Acknowledgements

The authors thank Larry Berdahl and Steve White for valuable suggestions. We are appreciative of the high quality routines MXM and MINV provided by Cray Research, Inc. This work was supported by the U.S.D.O.E. for the Lawrence Livermore National Laboratory under contract W-7405-ENG-48 and by the Association Euratom - Confederation Suisse at the Ecole Polytechnic Federale de Lausanne.

Figure 1: Matrix A_1 is block tridiagonal with dense blocks indicated by the shading.

Figure 2: The maximum possible multitasking overlap for the cyclic reduction algorithm is plotted versus the number of block rows.

Figure 3: Matrix A_2 is block tridiagonal with banded or sparse blocks indicated by the speckled pattern.

Figure 4: Matrix A_3 which contains four different kinds of blocks arises in some plasma stability models. The blocks are banded and often sparse.

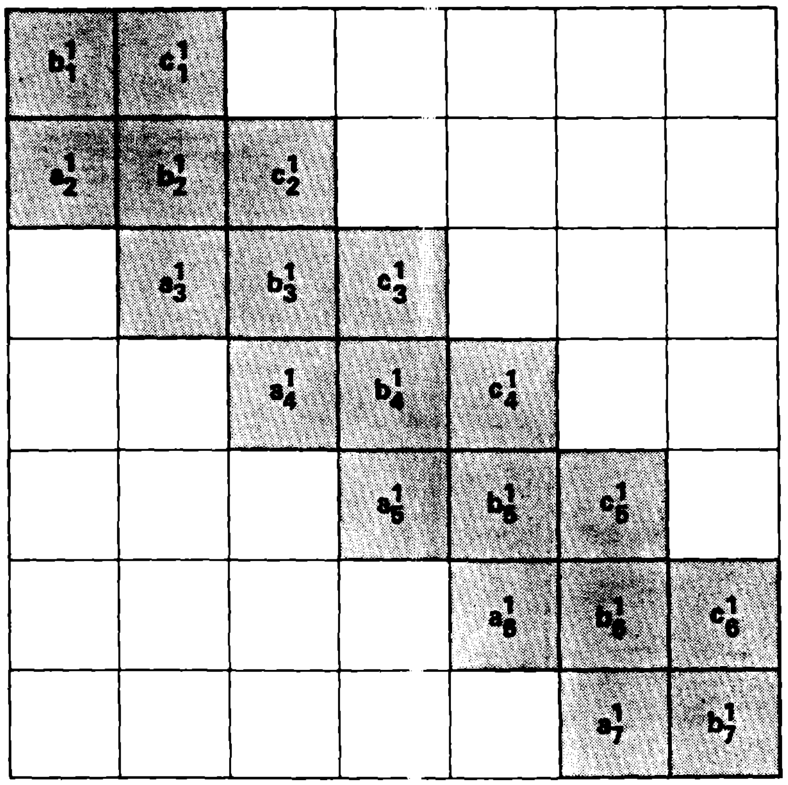
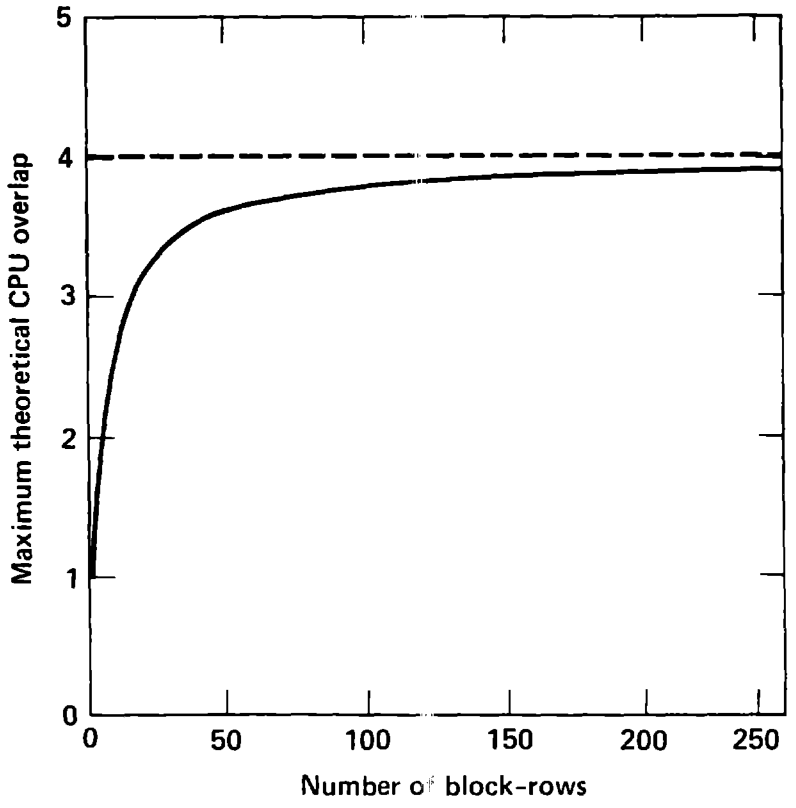

b_1^1
c_1^1
a_2^1
b_2^1
c_2^1
a_3^1
b_3^1
c_3^1
a_4^1
b_4^1
c_4^1
a_5^1
b_5^1
c_5^1
a_6^1
b_6^1
c_6^1
a_7^1
b_7^1

Figure 1



Maximum theoretical CPU overlap
5
4
3
2
1
0
0
50
100
150
200
250
Number of block-rows

Figure 2

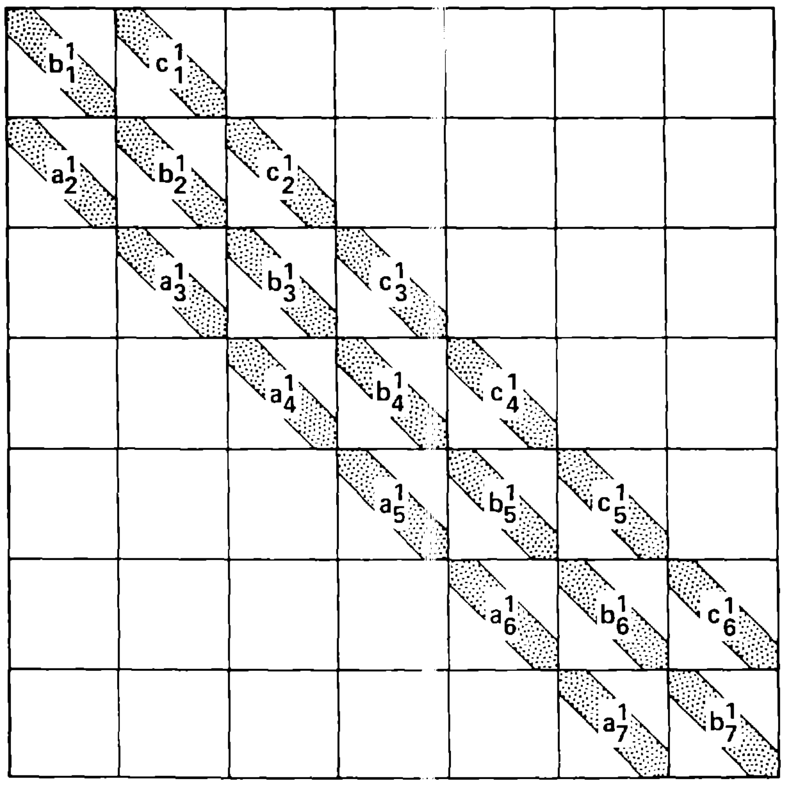
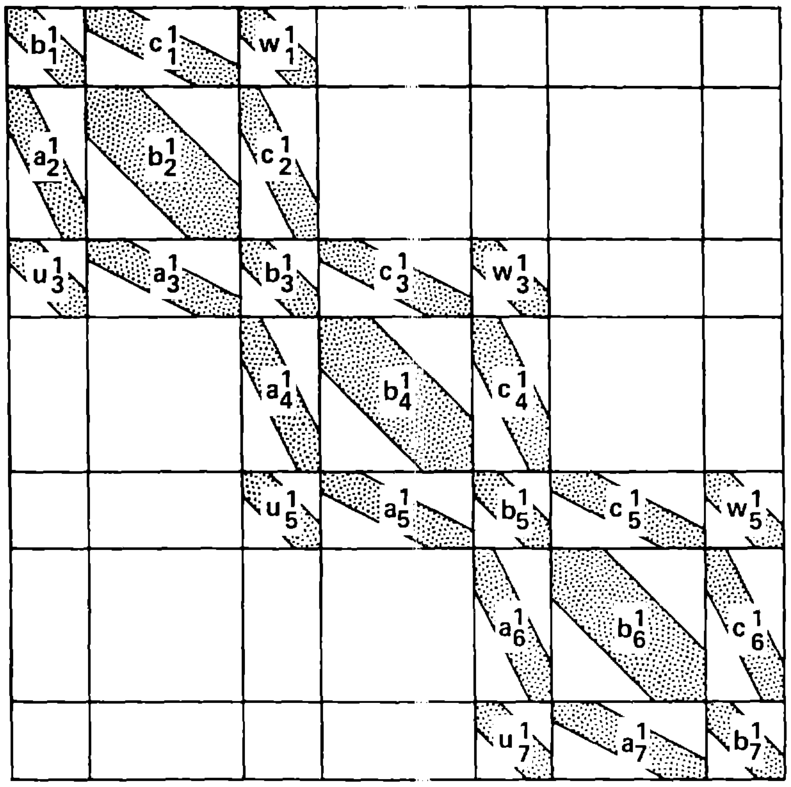

b_1^1
c_1^1
a_2^1
b_2^1
c_2^1
a_3^1
b_3^1
c_3^1
a_4^1
b_4^1
c_4^1
a_5^1
b_5^1
c_5^1
a_6^1
b_6^1
c_6^1
a_7^1
b_7^1

Figure 3



b_1^1
c_1^1
w_1^1
a_2^1
b_2^1
c_2^1
u_3^1
a_3^1
b_3^1
c_3^1
w_3^1
a_4^1
b_4^1
c_4^1
u_5^1
a_5^1
b_5^1
c_5^1
w_5^1
a_6^1
b_6^1
c_6^1
u_7^1
a_7^1
b_7^1

Figure 4